Augensummen von Würfeln – ein überraschend ertragreiches Thema

Ein triviales Ausgangsproblem und eine minimale Änderung

Eine der Standardaufgaben, die wohl schon jeder Informatiklehrer seinen Schülern im Rahmen der Behandlung von Schleifen gegeben hat, ist die folgende:

> *Schreibe ein Programm, das nach Eingabe einer Zahl s durch den Benutzer ermittelt, wie viele Möglichkeiten c es gibt, mit zwei Würfeln die Zahl s zu würfeln.*

Als Lösung erwartet man meist ein Programm mit zwei geschachtelten Schleifen und die Verwendung einer Sammler-Variable[1] (Prog.1).

```
int augsum(int s){
    int c,i,j;

    c=0;
    for(i=1;i<=6;i++){
        for(j=1;j<=6;j++){
            if(i+j==s) {c=c+1;}
        }
    }
    return(c);
}
```

Programm 1

Gerne stellt man nun zu schnellen Schülern noch die Aufgabe, das Gleiche mit drei oder vier Würfeln zu implementieren. Dies führ quasi zwangläufig zur Fragestellung nach w Würfeln und damit geradewegs in ein Problem: Diese Aufgabenstellung ist für die Schüler, gerade im Rahmen eines Anfängerkurses, in dem solche Aufgaben gestellt werden, zunächst nicht lösbar. Im Folgenden werden wir nun verschieden Methoden dieses Problem zu lösen betrachten und ihren didaktischen Wert kurz diskutieren.

Automatische Erzeugung von Code

```
function augsum(w,s) {
        programm="(function(){\n c=0;\n"
        for(i=0;i<w;i++){
            programm+="for(v"+i+"=1;v"+i+"<7;v"+i+"++){\n"
        }
        programm+="if(v0"
        for(i=1;i<w;i++){
            programm+="+v"+i;
        }
        programm+="=="+s+"){c=c+1;}\n"
        for(i=0;i<w;i++){
            programm+="}\n"
        }
        programm+="return c;})()";
        return(eval(programm));
}
```

Programm 2

Eine der ersten Ideen, die von Schülern vorgebracht wird, ist es, das Programm vom Computer selbst schreiben zu lassen. Wir demonstrieren dies hier am Beispiel der „Eval“-Funtion von JavaScript (Prog.2). Subsummiert man dieses Vorgehen unter dem Begriff „Selbstmodifizierender Code“, auch wenn in diesem Beispiel Code und „Meta-Code“ strikt getrennt bleiben, so öffnet sich ein breites Spektrum von Diskussionsmöglichkeiten, angefangen bei der Frage, warum dieses in früheren Zeiten verbreitete Konzept aufgegeben wurde bis hin zu moderner „programmierbarer Hardware“.

[1] Vgl. „Roles of Variables“, z.B. http://cs.joensuu.fi/~saja/var_roles/ (abgerufen am 4.4.2017)

Iteratives Aufzählen

Eine weitere „einfache“ Möglichkeit zur Lösung des Problems besteht in der iterativen Generierung aller Möglichkeiten. Hier stellen wir zwei Varianten vor, die beide auf das Zählen im Sechsersystem hinauslaufen. Dies kann entweder explizit geschehen (Prog. 3) oder durch Umrechnung einer Zahl in dieses System (Prog. 4).

```
function augsum(w,s:integer):integer;
var max,i,e,c,h,x:integer;
begin
 c:=0; max:=1;
 for i:=1 to w do max:=max*6;
 for i:=0 to max-1 do
 begin
    x:=0; h:=i;
    for e:=1 to w do
    begin
       x:=x+h mod 6+1;
       h:=h div 6;
    end;
    if x=s then c:=c+1;
 end;
augsum:=c;
end;
```

Programm 4

```
def augsum(w,s):
  c=0;
  d=[0];
  for i in range(w):
     d.append(1);
  while d[w]!=2:
     d[0]=d[0]+1;
     h=0;
     for i in range(w):
        if d[i]>6:
           d[i]=1
           d[i+1]=d[i+1]+1
        h=h+d[i]
     if h==s and d[w]!=2:
       c=c+1
  return c
```

Programm 3

Dieses naivste Vorgehen bietet einfache Einstiegsmöglichkeiten in die Thematiken, „vollständiges Aufzählen“, Laufzeitanalyse und „praktische Berechenbarkeit“, kann aber auch zur Wiederholung von Zahldarstellungen Anlass geben.

Rekursion

Mit $augsum(w,s) = \sum_{i=1}^{6} augsum(w-1, s-i)$,

$augsum(0,0) = 1$ und für

$s \in \mathbb{N}_+ : augsum(0,s) = 0, augsum(s,0) = 0$

erhält man eine leicht implementierbare Rekursionsgleichung (Prog. 5a). Kürze und Eleganz dieser Darstellung im Vergleich zu den vorangegangenen Programmen ist Werbung für und leichter Zugang zur Rekursion in einem. Die Aufteilung in w-1 und einen Würfel ist nicht zwingend, sondern nur der z=1 Spezialfall von:

```
public static long augsum(int w, int s)
{
      long c=0;
      if(s==0 && w==0){return 1;}
      if(s<=0 || w <=0 ){return 0;}
      for(int i=1;i<=6;i++) {
          c+=augsum(w-1,s-i);
      }
      return c;
}
```

Programm 5a

$augsum(w,s) = \sum_{i=z}^{6z} augsum(z,i) * augsum(w-z, s-i)$. Z.B. mit z=$\lfloor \frac{w}{2} \rfloor$ erhält man ein alternatives Schema (Prog. 5b), wobei man darauf achten muss, dass die Rekursion den Rekursionsanfang bei 0 ggf. nicht mehr erreicht. Dies gibt Anlass zur Diskussion der Effektivität verschiedener Rekursionsansätze.

```
public static long  augsum(int w, int s)
{
        long c=0;
        if(s==0 && w==0 || (w==1 && s>0 && s<7)){return 1;}
        if(s<=0 || w <=1){return 0;}
        for(int i=w/2;i<=s && i<=6*(w/2);i++){
                c+=augsum(w/2,i)*augsum(w-(w/2),s-i);
        }
        return c;
}
```

Programm 5b

Die unerfreuliche Laufzeit aufgrund von Mehrfachberechnungen derselben Werte führt unmittelbar zur Idee des nächsten Abschnitts:

Memoization

Während der Code des vorangegangenen Kapitels recht speichereffizient ist, werden entsprechend viele Werte mehrfach bestimmt. Eine einfache Möglichkeit zur – speicherintensiven – Beschleunigung bei seiteneffektfreien Funktionen besteht darin, einen Zwischenspeicher für die Funktionswerte anzulegen (Programm 6). Wegen ihrer universellen Anwendbarkeit und der nahezu demonstrativen Zeit/Platz Abwägung erscheint die Behandlung dieser Technik, „Memoization“ [2] genannt, besonders sinnvoll. Ebenso kann die verwandte Cache-Thematik hier eingebracht werden.

```
public static long[][] speicher;

public static long recurse(int w, int s){
    long c=0;
    if(s==0 && w==0){return 1;}
    if(s<=0 || w <=0){return 0;}
    if (speicher[w][s]!=-1){return speicher[w][s];}
    for(int i=1;i<=6;i++){
        c+=recurse(w-1,s-i);
    }
    speicher[w][s]=c;
    return c;
}

public static long augsum(int w, int s){
    speicher=new long[w+1][s+1];
    for(int i=1;i<=w;i++){
        for (int e=1;e<=s;e++){speicher[i][e]=- 1;}
    }
    return(recurse(w,s));
}
```

Programm 6

Branch&Bound

Offensichtlich kann es keine Lösungen mehr geben, wenn die Zahl der Würfel größer ist als die verbleibende Augensumme (Minimum pro Würfel ist 1) oder wenn die verbleibende Augensumme größer ist als die Zahl der Würfel mal 6. Diese Äste der Rekursion kann man sofort abschneiden. Im Gegensatz zum „echten Branch&Bound“ werden hier keine adaptiven oder globalen oberen und unteren Schranken verwendet, die Ähnlichkeit in der Überlegung lässt dies aber als geeignete Propädeutik erscheinen. Es ändert sich gegenüber Programm 6 nur eine Zeile (6a):

```
if(s<=0 || w <=0 || w>s || w*6<s){return 0;}
```

Programmstück 6a

Dynamische Programmierung

Statt die Speicherung der Zwischenwerte dem Stack oder dem Zwischenspeicher zu überlassen kann man diese natürlich auch direkt verwalten (Prog. 7) und erhält durch verbesserte Kontrolle über die Einzelschritte die Möglichkeit zu weiterer Optimierung. Die einfachste, auch für Programmierunerfahrene geeignete Variante besteht darin, auszunutzen, dass sich die Rekursionsgleichung auf aufeinanderfolgende

```
def augsum(w,s):
  d=[]
  x=[1]
  d.append(x)
  for i in range(w):
    x=[]
    for e in range(6*(i+1)+1):
      c=0
      for j in range(1,7):
        if(e-j>=0 and e-j<=6*i):
          c=c+d[i][e-j]
      x.append(c)
    d.append(x)
  return x[s]
```

Programm 7

[2] https://de.wikipedia.org/wiki/Memoisation (Abgerufen am 19.1.2017)

Vorgänger bezieht und eine Tabellenkalkulation zu verwenden. Da diese leider mit negativen Zellbezügen nichts anfangen kann, verschieben wir das Ganze aber um 7 Felder und tragen die Möglichkeiten, negative Zahlen zu würfeln, entsprechend mit 0 ein. In Feld A6 vermerken wir die Möglichkeit, die Zahl 0 mit 0 Würfeln zu würfeln, mit 1, in Feld B7 berechnen wir die Möglichkeit, mit einem Würfel die Zahl 1 zu würfeln, als B7= SUMME(A1:A6).

Nun können wir durch die „Füllbereich“ („aufziehen“) Funktion und die damit verbundene relative Interpretierung der Zellbezüge die weitere Tabelle automatisch erzeugen lassen (Tab 1).

```
def augsum(w,s):
    d=[1]
    for i in range(w):
        x=[0]
        c=0
        for e in range(6*(i+1)):
            if(e<=6*i):
                c=c+d[e]
            if(e-6>=0):
                c=c-d[e-6]
            x.append(c)
        d=x
    return x[s]
```

Programm 8

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0
8	0	1	1	0	0	0	0	0	0	0
9	0	1	2	1	0	0	0	0	0	0
10	0	1	3	3	1	0	0	0	0	0
11	0	1	4	6	4	1	0	0	0	0
12	0	1	5	10	10	5	1	0	0	0
13	0	0	6	15	20	15	6	1	0	0
14	0	0	5	21	35	35	21	7	1	0
15	0	0	4	25	56	70	56	28	8	1
16	0	0	3	27	80	126	126	84	36	9
17	0	0	2	27	104	205	252	210	120	45
18	0	0	1	25	125	305	456	462	330	165
19	0	0	0	21	140	420	756	917	792	495
20	0	0	0	15	146	540	1161	1667	1708	1287
21	0	0	0	10	140	651	1666	2807	3368	2994
22	0	0	0	6	125	735	2247	4417	6147	6354

Tabelle 1

Schnell fällt auf, dass jede Spalte in der Tabelle bzw. jedes Array x im obigen Programmcode nur von der/dem jeweils vorangehenden abhängt. Es ist also nicht nötig, die weiter zurückliegenden Ergebnisse zu speichern, und man kann beispielsweise in Python deren Entfernung der Garbage-Collection überlassen (Prog.8; d=x entfernt das „alte“ d); in anderen Sprachen kann man zwei Arrays alternierend verwenden. Eine weitere Optimierungsmöglichkeit ergibt sich aus der Erkenntnis, dass die jeweils zu bestimmenden Summen sich nur um zwei Summanden unterscheiden: Ein neuer (letzter) tritt hinzu, der bisher erste fällt weg. Bei dieser Technik lassen sich Laufzeit und Codeoptimierungen sichtbar leichter behandeln als bei Rekursion.

Erzeugende Funktionen

Eine einigermaßen geschlossene Lösung des Problems erhält man über erzeugende Funktionen; diese Thematik liegt aber – auch wenn hier keine Funktionalgleichungen auftreten – am Rand des schulischen Horizonts. Der Abrundung halber geben wir hier kurz die Herleitung in Anlehnung an http://mathworld.wolfram.com/Dice.html (abgerufen am 6.9.2016) an:

Multipliziert man den Term $(x^6 + x^5 + x^4 + x^3 + x^2 + x^1)$ mit sich selbst, so erhält man

$(1x^{12} + 2x^{11} + 3x^{10} + 4x^9 + 5x^8 + 6x^7 + 5x^6 + 4x^5 + 3x^4 + 2x^3 + 1x^2 + 0x^1)$; die Koeffizienten entsprechen dabei der Zahl der Möglichkeiten, wie sich der Exponent aus zwei Zahlen von 1 bis 6 zusammensetzen lässt. Entsprechend erhält man die Anzahl wie oft sich s als Summe von w Würfeln darstellen lässt als den s ten Koeffizienten von $(x^6 + x^5 + x^4 + x^3 + x^2 + x^1)^w$. Um diesen zu bestimmen, bringt man diesen Ausdruck in eine Reihendarstellung:

$$x^w\left(\sum_{i=0}^{5} x^i\right)^w = x^w\left(\frac{1-x^6}{1-x}\right)^w = x^w(1-x^6)^w\left(\frac{1}{(1-x)}\right)^w = x^w\left(\sum_{i=0}^{w}(-1)^i * \binom{w}{i} x^{6i}\right)\left(\sum_{j=0}^{\infty} x^j\right)^w$$

Um $\left(\sum_{j=0}^{\infty} x^j\right)^w$ weiter aufzulösen macht man sich nun analog klar, dass die Koeffizienten der resultierenden Potenzreihe der Zahl der Möglichkeiten entsprechen, den jeweiligen Koeffizienten k als Summe von w beliebigen Zahlen (incl. 0) zu schreiben. Lässt man zunächst die Null weg, so entspräche dies der Vorstellung, ein Objekt der Länge k an $w-1$ ganzzahligen Stellen in w Teilobjekte zu zerschneiden, wozu es $\binom{k-1}{w-1}$ Möglichkeiten gibt. Zur Berücksichtigung der Null greift man dann zu dem Trick, stattdessen ein Objekt der Länge $k+w$ in w Stücke zu zerschneiden und danach jedes Stück um 1 zu kürzen, was insgesamt wieder k ergibt, wobei dann Stücke der Länge 0 entstehen können. Somit erhält man $\binom{k+w-1}{w-1}$ bzw. wegen der Symmetrie in der Definition des Binomialkoeffizienten $\binom{k+w-1}{k}$. Eingesetzt in den obigen Term erhält man

```
def augsum(w,s):
  bino2=[1]
  h=1
  for r in range(1,s-(w-1)):
    h=h*(w+r-1)/r
    if ((s-r-w)%6 == 0):
      bino2.append(h)
  sum=0
  einspm=1
  wueberi=1
  for i in range((s-w)//6+1):
   sum=sum+einspm*wueberi*bino2[len(bino2)-1-i]
   einspm=-einspm
   wueberi=wueberi*(w-i)/(i+1)
  return sum
```

Programm 9

$x^w \sum_{i=0}^{w}(-1)^i \binom{w}{i} x^{6i} \sum_{k=0}^{\infty} \binom{k+w-1}{k} x^k$. Zum s ten Koeffizienten dieser Reihe tragen alle Summanden mit $w+6i+k=s$, also $k=s-6i-w$ bei und man erhält somit für die Anzahl der Möglichkeiten mit w Würfeln eine Augensumme von s zu erreichen:

$$augsum(w,s) = \sum_{i=0}^{s}(-1)^i \binom{w}{i}\binom{s-6i-1}{s-6i-w} = \sum_{i=0}^{s}(-1)^i \binom{w}{i}\binom{s-6i-1}{w-1}.$$

Da $s-6i-w$ nur für $i < \frac{s-w}{6}$ positiv ist, lassen sich noch ein paar Summanden entfernen zu $augsum(w,s) = \sum_{i=0}^{\lfloor\frac{s-w}{6}\rfloor}(-1)^i \binom{w}{i}\binom{s-6i-1}{w-1}$.

Die Werte von $(-1)^i$ – einspm - und $\binom{w}{i}$ - wueberi - können effizient aus denen der jeweils vorangegangenen Iteration bestimmt werden. Bei der Berechnung von $\binom{s-6i-1}{w-1}$ erschien es geschickter, mit $r := 0$ und $h := \binom{w-1+r}{w-1}$=1 zu beginnen und dann diejenigen Werte herauszupicken, für die (s-1)-(w-1+r) =6 i durch 6 teilbar ist und diese dann in umgekehrter Reihenfolge einzusetzen.

Es wird – trotz sprechender Variablennamen – wohl niemand Programm 9 noch ansehen, was es berechnet. Seine Behandlung wäre wohl eher im Bereich Kombinatorik in einem Mathematik-Leistungskurs zu empfehlen.

OStR. Dr. Martin Löhnertz
Universität Trier/St.-Willibrord-Gymnasium Bitburg
H526 Behringstraße 21
D-54296 Trier
E-Mail: public@loehnertz.de